EMDAT – DEVELOPER MANUAL

Samad Kardan, Sébastien Lallé, Dereck Toker, and Cristina Conati. 2021. EMDAT: Eye Movement Data Analysis Toolkit (2.x). The University of British Columbia. DOI: 10.5281/zenodo.4699774

Manual generated by Shunsuke Ishige and Sébastien Lallé on August 4, 2017.
Latest update: June 23, 2021.

Introduction

The aim of this manual is to elicit the EMDAT source code for developers. Please refer to the User Manual if you solely want to use/apply EMDAT.

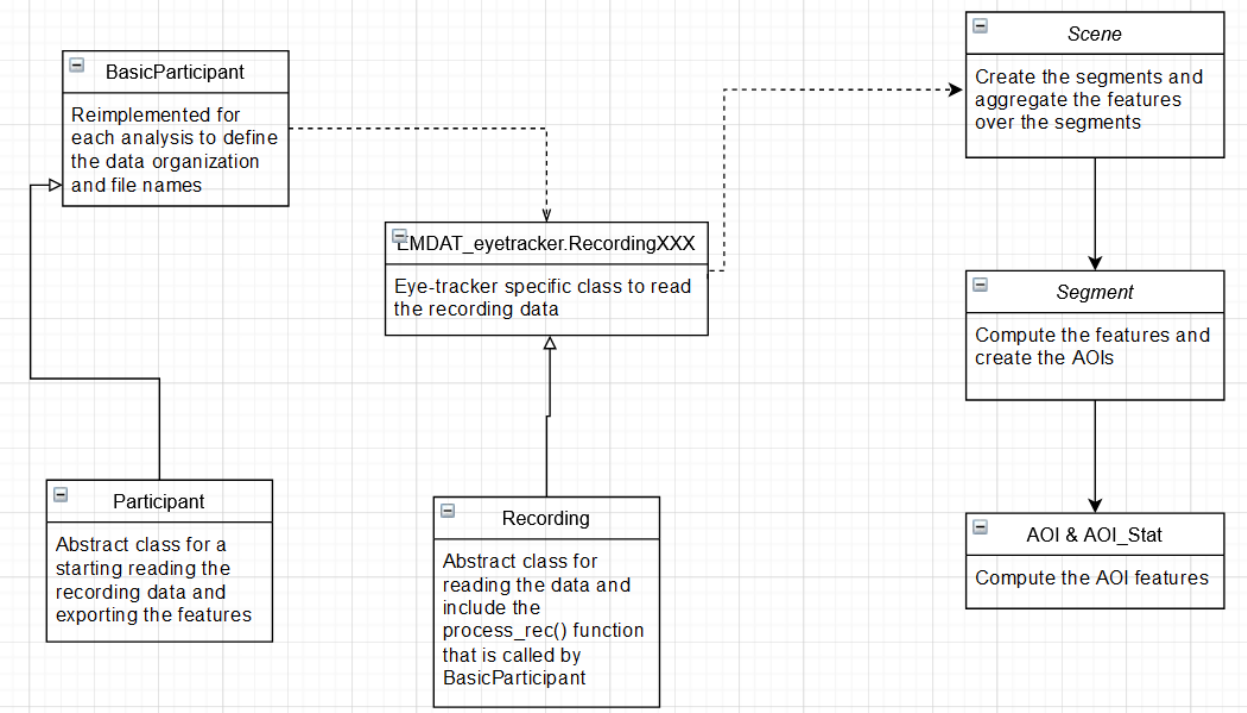
To lay out the plan for the manual, we first track the general execution flow, through which input eye-tracking data (we will use TobiiV3 data as an example throughout the manual) are aggregated into EMDAT output data, with little mention of the data transformation itself. Subsequently, we discuss how the raw input data (Tobii export file) are processed first into lists of preprocessed data (list of Fixations, that of Saccades, and so on), then into the attribute values of Segment objects, and finally into the attribute values of Scene objects. The rationale for this presentation order is that the latter takes place over the course of successive method calls involving several different class objects and, as a result, prior understanding of the former should facilitate following the subsequent discussion of data transformation per se. Once the basic aggregation is explained, against that background knowledge we discuss the AOI processing, which is actually part of the larger aggregation process involving scenes and segments.

The last part of the manual shows how to perform the main developer tasks with EMDAT, namely adding a new eye-tracking data format, and adding new features.

Overview of the class structure

EMDAT's class structure is shown in the next Figure. On this figure:

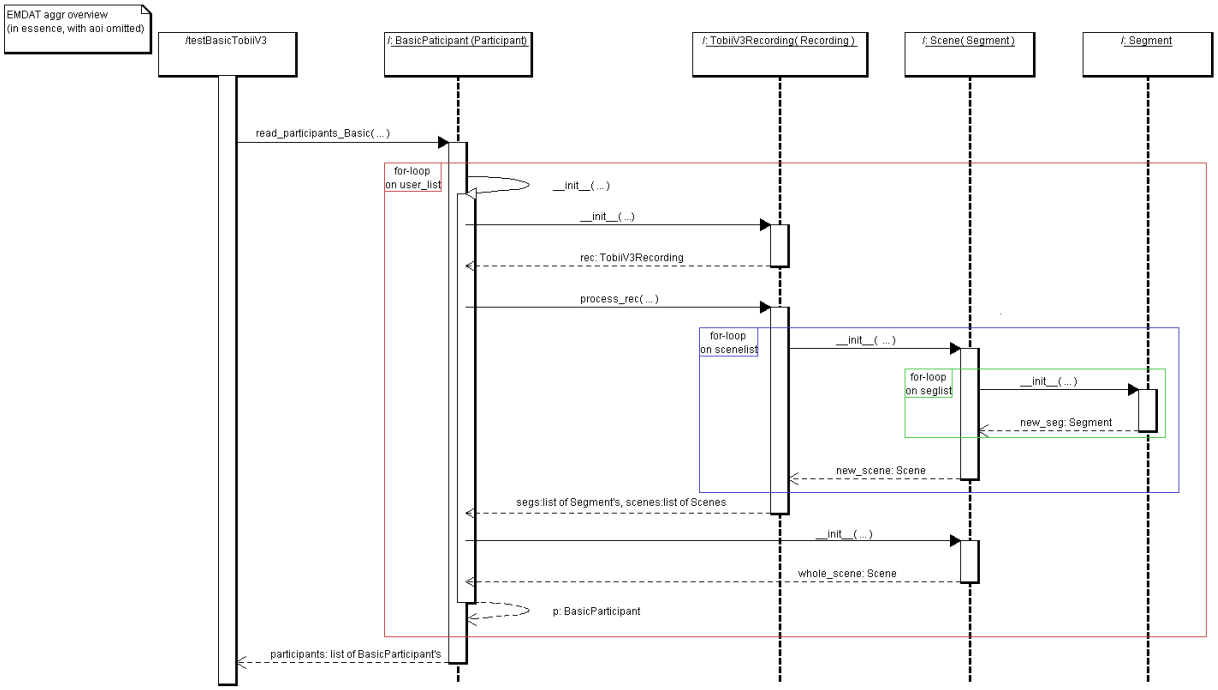
- The classes on the right (Scene, Segment, and AOI) are meant to compute the EMDAT's feature. Namely the AOI class measure the AOI features within a segment. The Segment class compute in addition the non-AOI features. The Scene class aggregate the features (both AOI-based and non-AOI-based) over the segments that compose the class. Refer to the User Manual (Section 4) for the definition of scenes and segments.
- The classes in the middle (Recording…) are meant to read the data for each recording. Because EMDAT can support different eye-tracking data format (cf. User Manual, Section 4), the Recording class is instantiated for each of the supported format in the EMDAT_eyetracker folder
- The Participant class includes the functionality to start processing the recording and exporting the features. The BasicParticipant class is meant to be experiment-specific so that the user can fine-tune reading of the recording depending on the file architecture, file names, file location… (See User Manual, Section 5).



(Figure0: EMDAT classes)

An Overview of EMDAT Execution Flow

Figure1 shows how the classes interact with each other's as part of the main execution flow.



(Figure1: EMDAT aggregation overview)

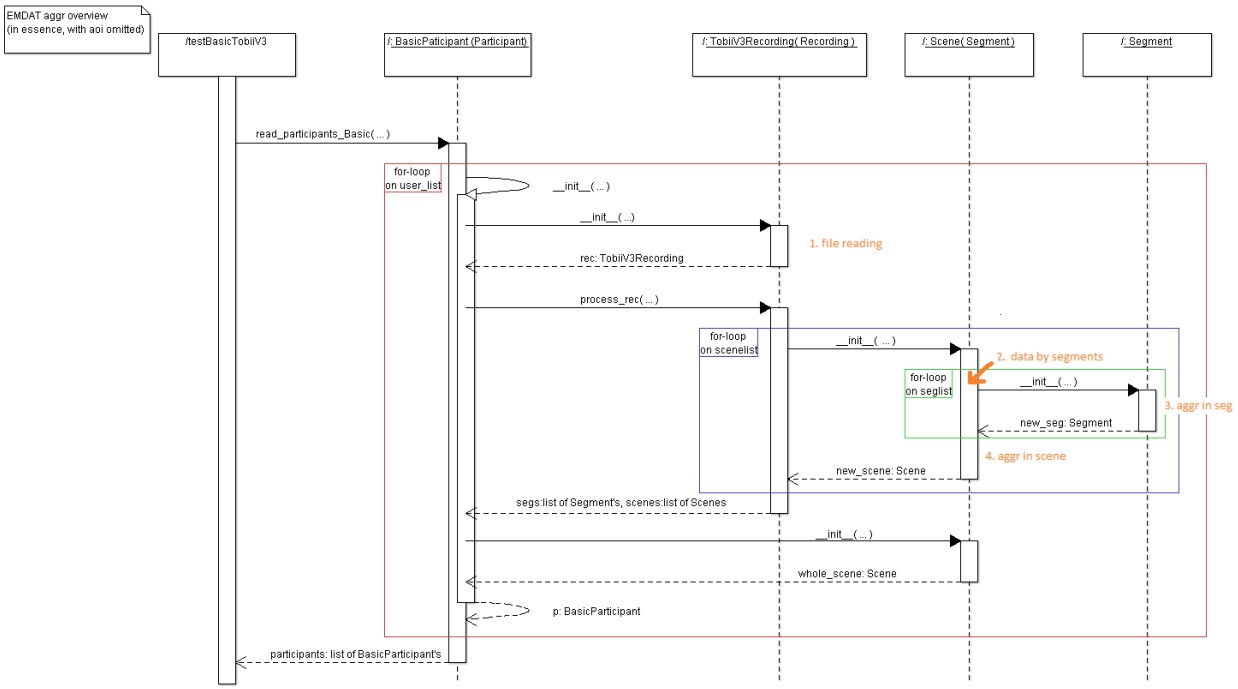
In essence, the EMDAT execution flow may be represented as in Figure1 above: the central part consists of the nested loop straddling several different class objects. Inevitably some details are omitted, but this representation should be robust enough; the reader could add necessary details for other functionalities later as desired. Perhaps the easiest way to explain how this complex loop works, as the EMDAT processes given participants with their respective Tobii data and session information, would be to use actual examples and track the execution, referring to the above figure. To this end, let us use Recordings P17 and P18 from the TobiiV3 sampledata folder. The former has two scenes, namely part1 and part 2, each of which consists of a single segment, also called part1 and part2, respectively. The later has only one scene, main_task, but that comprises two segments called part1 and part2. The execution flows as follows.

1. The outer for-loop (shown in red in the figure) begins when the call to a method of BasicParticipant, read_participants_Basic, is made (in this case from the script testBasicTobiiV3.py).
2. The loop starts on P17. In particular, as in the figure, there is a call to the constructor of BasicParticipant in read_participants_Basic to create an object encapsulating the data for the participant.
3. In the constructor of BasicParticipant, a call to the constructor of TobiiV3recording is made for preprocessing of the Tobii data for the participant, which is to be discussed in detail in the next section.
4. A call to the method process_rec made on the TobiiV3Recording object just created, in turn, leads to the middle loop (shown in blue in the figure) that creates Scene objects for the scenes belonging to the participant, namely part1 and part2.
5. The middle loop runs first on the scene part1, which, in turn, commences the inner loop (shown in green in the figure) to create a Segment object, during which the preprocessed Tobii data is further processed and stored in the attribute values of the Segment object. Recall that there is (segment) part1 for (scene) part1. Since the scene part1 has only one segment, the inner loop runs only once.
6. Generally, when the iteration of the inner loop (green one) ends, with all Segment objects being created, the data encapsulated in the Segment objects are further aggregated in the Scene constructor and stored in the attribute values of the Scene object.
7. The middle loop (blue one) runs again for the scene part2. So, by the time the call to process_rec returns, the Tobii data belonging to P17 have been aggregated and stored by scenes.
8. Another call to the constructor of Scene is made, passing all Segment objects created for this participant in the previous steps as an argument; this is for finding aggregated data for the participant as a whole, rather than for individual scenes. (The data so computed populate the 17_allsc row of the EMDAT output file.)

9. The outer loop (red one) runs now on P18. The process is essentially the same as described above, with one exception: since P18's only scene main_task consists of two segments, part1 and part2, the inner loop (green one) runs twice while the middle loop (blue one) only once.
10. Finally, although this is not shown in Figure 1, a call to a file export method is made from the script testTobiiV3Recording, rendering certain attribute values of the Scene objects and those of the whole_scene objects of the type Scene (created as in step 8, one for P17 and another for P18) into the rows of the EMDAT output file.

Data Transformation and Non-AOI Feature Generation

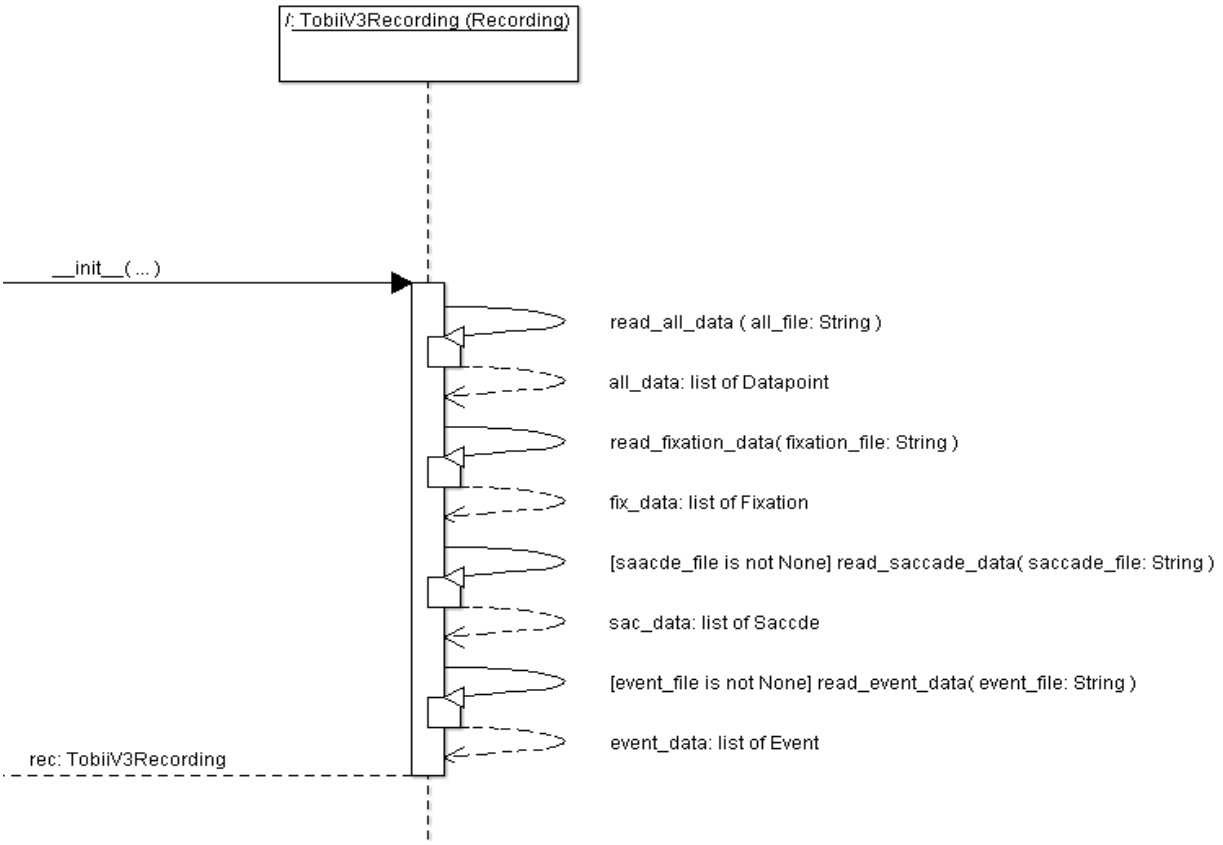
In this section, I explain how the raw input Tobii data are transformed as the execution flows, in such a manner as to relate the discussion to the overall view presented above. In particular, here is the same figure with the major events in data transformation marked in orange. I elaborate them in the order indicated. To reiterate, the approach is not to exhaustively describe but to concisely convey the characteristics of the EMDAT code, providing a framework for comprehension and organization of details.



(Figure 2: EMDAT aggregation overview with data transformation events indicated)

File reading (1 in Figure 2): TobiiV3Recording is initialized, during which process the Tobii data for the participant is read from the file system of the machine and transformed into lists of preprocessed data, which are later passed as arguments of the constructor of Scene for the subsequent data processing. To explicate this rather condensed description, let us first take a look at the following diagram, which is

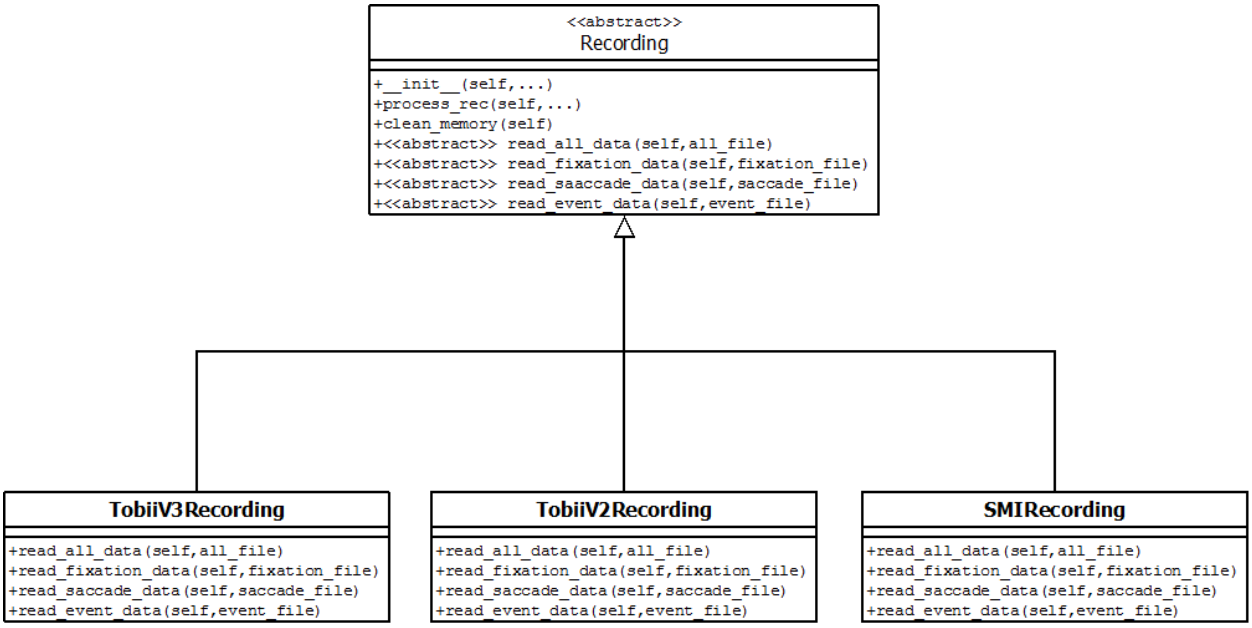
actually part of the overview figure above (in particular, the initialization of TobiiV3Recording) but has additional details related to data transformation explicitly shown.



(Figure 3: a more detailed view of TobiiV3Recording initialization)

The implementation details of these file read methods should not concern us given our goal of high-level understanding; what they do is essentially as follows. They open the file, whose path is specified by the string argument, and iterate over the rows of the input data file, recording only the values of certain columns (which depend on each read method and there may be some computational processing in some cases) of those rows that satisfy given conditions (again which depend on each read method) into dictionary objects. To store the extracted data for later use, the dictionary objects are converted into EMDAT class objects by passing the former into the respective initialization methods of the classes, Datapoint, Fixation, Saccade, and Event. Those lists are stored as attribute values of the instantiated TobiiV3Recording; that is, the value assignment of the form self.some_data = read_some_ data (…) takes place. Notes: (1) dictionary is a Python data structure of the form { key1: value1, … }; (2) in our case, namely TobiiV3Recording, the file read by all those read methods is the same Tobii data file of the participant, in spite of the different parameter names.

Although the reader can skip this paragraph, it might be worth mentioning here that there is a subtlety related to the class inheritance.



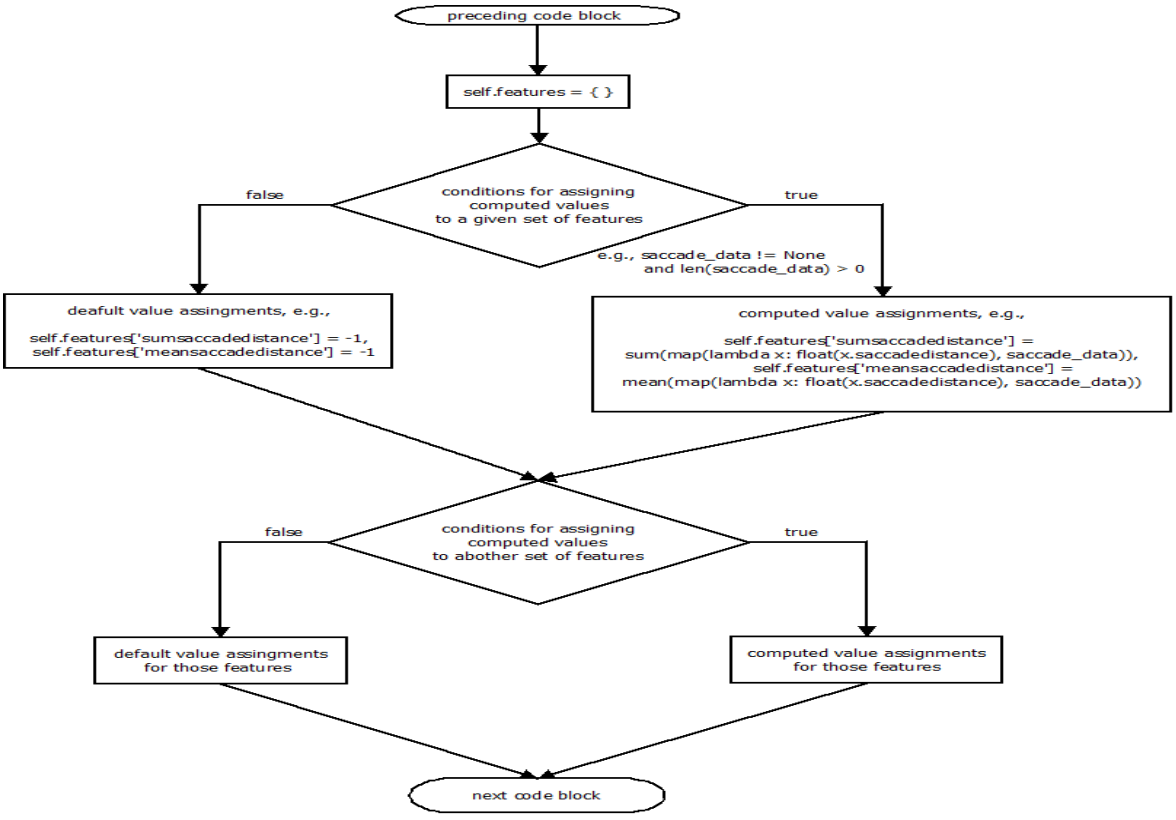
(Figure 4: Recording class inheritance. Attributes omitted)

As in Figure 4, TobiiV3Recording extends the abstract class Recording. In particular, it inherits the initialization method, so that when a call to instantiate TobiiV3 happens as in Figure 3, the calls to the read methods during the instantiation are actually passed to the read methods as implemented in TobiiV3Recording through class inheritance. The same observation may be made for the other two subclasses, if they are used instead of TobiiV3Recording. Note, however, that all three subclasses inherit the same implementation of the method process_rec from their parent class, so that the call to that method after instantiation of a Recording subclass (TobiiV3 in Figure 2) is not specific to the subclass. The implication is that the overview should still apply even if TobiiV3 is replaced with another subclass.

Sub-listing of data by segments (2 in Figure 2): Since the lists of preprocessed data from above contain all data for the entire session of the participant, possibly involving multiple time intervals, sub-listing of them by using the session information in the seg file takes place before the data aggregation at the segment level. Obviously, there are two components here, namely those data lists and the session information. To explain how the former reach the point 2, first, recall from the preceding discussion that those lists of EMDAT class objects are stored as the attributes of the TobiiV3 object; those values are simply passed as arguments of the Scene initialization method that precedes the point 2 in Figure 2. Now, as for the session information, there is actually a call to a method for reading and processing the seg file, stored in the machine, in BasicParticipant between the return of the call to the TobiiV3 constructor and the beginning of the call to process_rec, although it is not shown in Figure 2 (the reason is two-fold: limitations of ArgoUML and the fact that the EMDAT also allows processing of the seg file in process_rec). The effect of calling the method, partition, is to store the time intervals recorded in the seg file in a dictionary of list values, which is subsequently used for sub-listing the data lists at the point 2. To see how the stored session information fits in the execution in Figure 2, let us use MapQuest P17 and P18 for examples again.

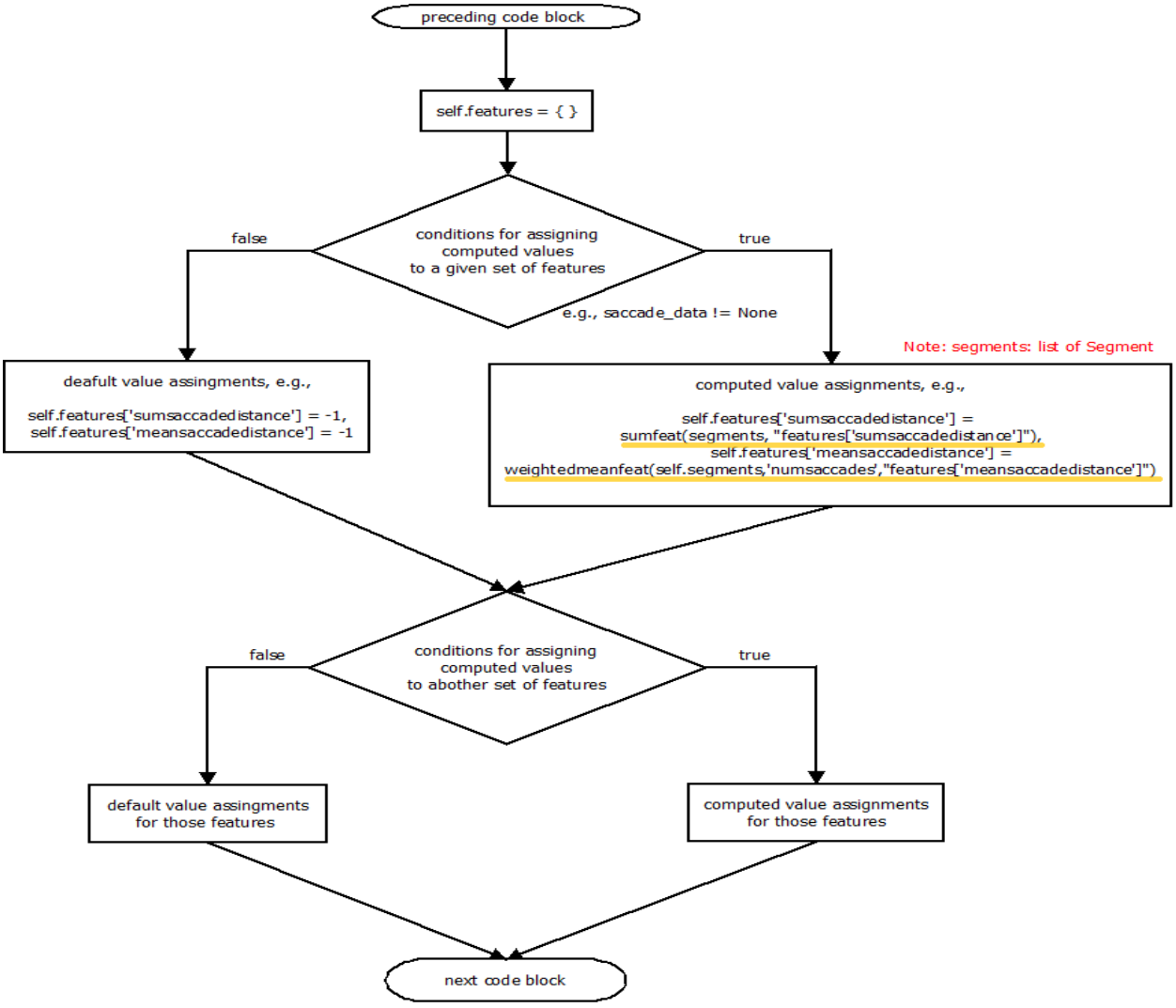
For P17, with two scenes, part1 and part2, each of which has one segment, also called part1 and part 2, respectively, the returned dictionary (scenelist in Figure 2) is as follows: {'part1': [('part1', 72727, 158389)], 'part2': [('part2', 158389, 367172)]}. Recall from the discussion in the first section that the middle for-loop (the blue one in Figure 2) is for the scenes of the participant. When that for-loop starts to run on the scene part1, the list (seglist in Figure 2) [('part1', 72727, 158389)] is passed to the Scene constructor as the segment belonging to the scene. The time interval starting at 72727 and ending at 158389 is then used for delimiting the lists of preprocessed data at the point 2 in Figure 2; similarly for the second iteration of the middle loop for the scene part 2. (Note: finding the starting and ending times for each list of preprocessed data actually happens through a function, using those times in the tuples as arguments.) Now, for P18, the dictionary is as follows: {'main_task': [('part1', 66808, 146471), ('part2', 146471, 324262)]}. So in this case, when the middle loop runs on the scene main_task, [('part1', 66808, 146471), ('part2', 146471, 324262)] for the two segments is passed to the Scene constructor. In particular, for each iteration of the inner loop (first for segment part1 and then for segment part2), the respective time interval delimits the lists of preprocessed data.

Segment level aggregation (3 in Figure 2): Once the Segment constructor is called, the sublists of preprocessed data for the segment, which are passed through arguments of the method, are further processed, and the computed values are stored in the attributes of the object. In particular, the values that are eventually exported as the EMDAT aggregation output is assigned to the object's dictionary attribute called features, essentially according to the following schematically represented logic:



(Figure 5: schematic representation of aggregation in Segment __init__)

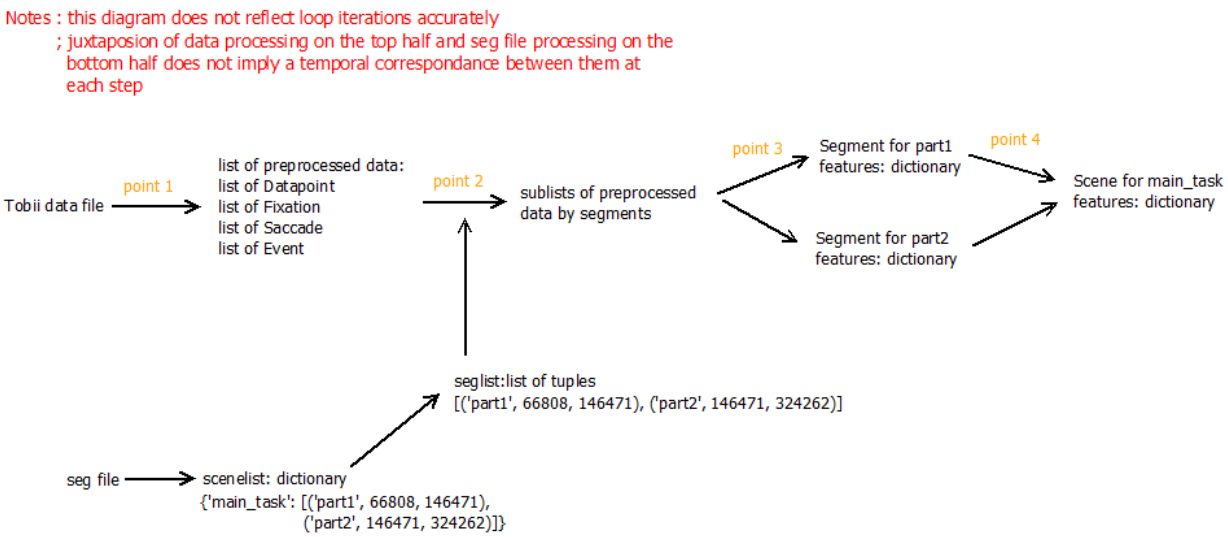
Scene level aggregation (4 in Figure2): When the inner for-loop (the green one in Figure 2) terminates with all Segment objects for the given scene being instantiated, a higher level of data compilation happens in the Scene initialization method: attribute values of those Segment objects are now aggregated. The general pattern of aggregation here, however, may also be conveyed by the same schematic representation of the processing logic as above:



(Figure 6: schematic representation of aggregation in Scene __init__)

Note, however, that as illustrated by the highlighted lines in Figure 6, the computation of a given value in the Scene object's dictionary attribute, also called features, is now made by aggregating over the corresponding values in all Segment objects' dictionary attributes of the same name.

To summarize the data flow explained so far in terms of a figure for P18:

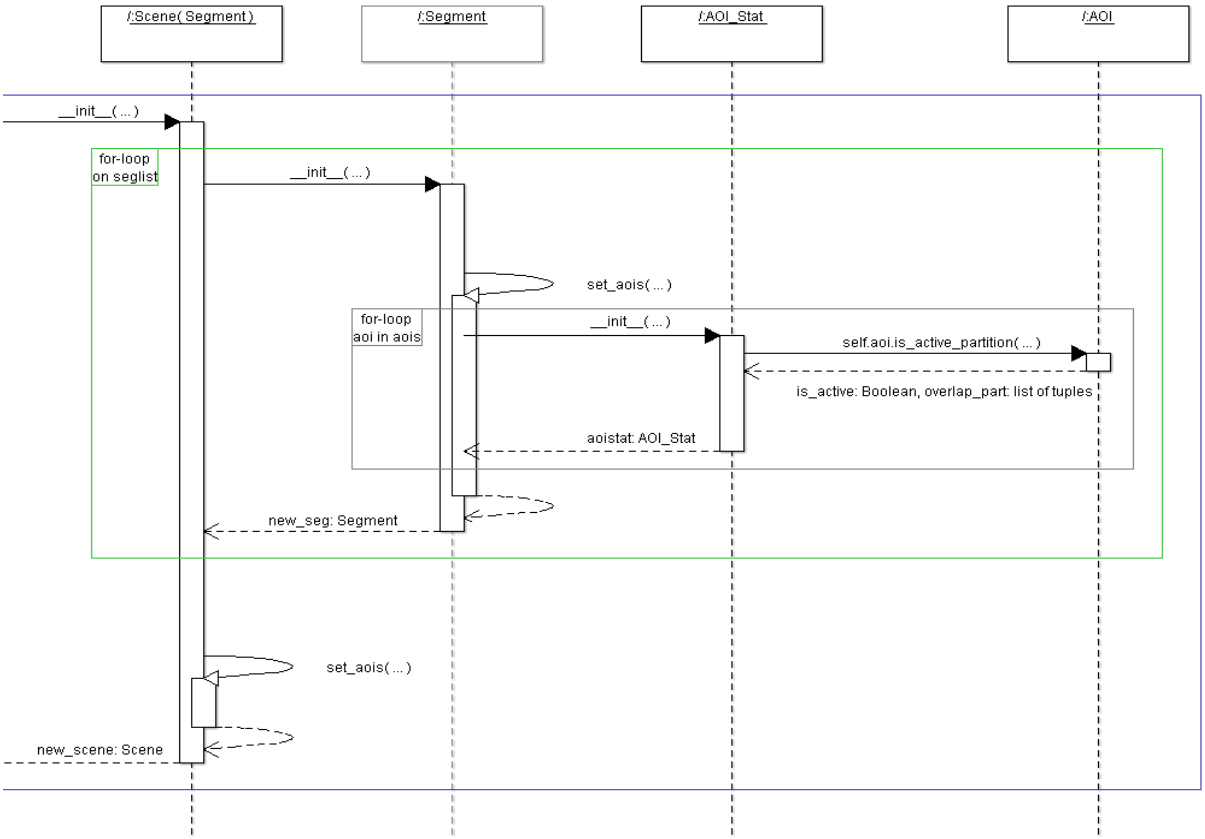


(Figure 7: data flow summary)

AOI Aggregation

It may be said that in terms of the code, static AOI's are a subset of dynamic ones, so that I discuss AOI aggregation at the greater generality. As I said earlier, AOI aggregation is actually part of the larger aggregation process that takes place at the segment and scene levels. Accordingly, in the figure below for an overview of AOI aggregation execution flow, I use the same coloring scheme as for Figure 1, with blue for the scene level middle loop and green for the segment level inner loop (Note: I omitted the participant level outer loop, the red one, in this figure for AOI aggregation.) As you can see in Figure 8 below, there is an additional loop for AOI's, marked in grey. The reader should try to mentally fit this more detailed pictorial representation into the Figure 1 for a comprehensive overview of EMDAT.

A few notes on the manual in this section are in order. In the preceding discussion, I separated the description of the execution flow from that of data transformation. However, by this time, the reader should be familiar with my representation scheme; I present the material here by discussing execution flow and data transformation simultaneously. Also, to relate this section on AOI aggregation to the preceding sections on the larger aggregation process, I use the TobiiV3 P18 sample recording as an example again when illustration is required.



(Figure 8: an overview of AOI aggregation execution flow)

Now, as you can see in Figure 8, the AOI level loop runs on the object called aois. To explain first what this is and where it comes from, although I omitted it in Figure 1 earlier, there is actually a call to a function (defined in Recording.py) in the initialization method of BasicPaticipant.py between the call to the constructor of TobiiV3 and the call to the method process_rec. The function reads the AOI definitions from the file system of the machine. For each AOI, it creates an AOI object, storing the AOI information in the attributes. The function returns a list of AOI objects, namely aois. The list aois is successively passed as an argument of the pertinent methods, to be used in the AOI level loop marked in grey in Figure 8: process_rec, the initialization methods of Scene and Segment, and finally the set_aoi method in Segment.

In explaining this looping on aois, I first summarize what takes place in each iteration in general terms and then illustrate the processes with MetroQuest P18 in such a manner as to put the discussion in the larger context of loops outside. In the subsequent discussion, I refer to these processes by the numbering symbols in bold.

LP1. The start and end times as well as fixation_data and event_data of the Segment object, in which the call to the constructor of the class AOI_Stat takes place, is passed to the initialization method of AOI_Stat.

LP2. The method is_active_partition is called on the AOI object. The returned Boolean is_active indicates whether the AOI is deemed active; that is, generally whether there is any intersection between the time interval determined by the Segment's start and end times on the one hand and the interval(s) of the AOI object stored in its timeseq attribute on the other. There are two special cases: an AOI with an empty interval, i.e., timeseq: [] is regarded as always active; and an AOI with negative 1's as in timeseq: [(-1, -1)] is considered as always inactive. (Note the former accounts for the fact that static AOI's are a subset of dynamic AOI's in terms of the code: static AOI's have an empty time interval.)

LP3. After the call returns to the AOI_Stat object, data transformation takes place on the fixation_data and event_data. If the AOI is deemed inactive according to the value of is_active, the default values are assigned to the attributes of the AOI_Stat object and the call returns to set_aois in the Segment object; otherwise, assignment of aggregated values may take place in the initialization of the AOI_Stat object. The aggregation may not involve all data in the fixation_data and event_data: those two lists are generally delimited with respect to time and space. The time is determined by intersections of intervals mentioned in LP2, and the space, by the area specified by the AOI object's polyin and polyout attribute values. Only the latter is relevant, however, for the second of the two special cases mentioned in LP2 above.

LP4. When the call returns to set_aoi, the returned value, i.e., the newly created AOI_Stat object is stored in the dictionary attribute of the Segment object, self.aoi_data, using the aid attribute value of the object as the key.

In the following illustration of the above summary with MetroQuest P18, let us consider AOI's that divide the screen into quadrants, each of which has its own active time intervals. The elements, i.e., AOI objects of the list aois returned by the function mentioned above look like as follows in the dictionary format:

```
{'aid': '1',
 'timeseq': [],
 'polyin': [(1, 1), (639, 1), (639, 511), (1, 511)],
 'polyout': []
}
{'aid': '2',
 'timeseq': [(100000, 150000)],
 'polyin': [(640, 1), (1279, 1), (1279, 511), (640, 511)],
 'polyout': []
}
{'aid': '3',
 'timeseq': [(-1, -1)],
 'polyin': [(1, 512), (639, 512), (639, 1023), (1, 1023)],
 'polyout': []
}
{'aid': '4',
 'timeseq': [(50000, 55000), (120000, 122000), (500000, 700000)],
 'polyin': [(640, 512), (1279, 512), (1279, 1023), (640, 1023)],
 'polyout': []
}
```

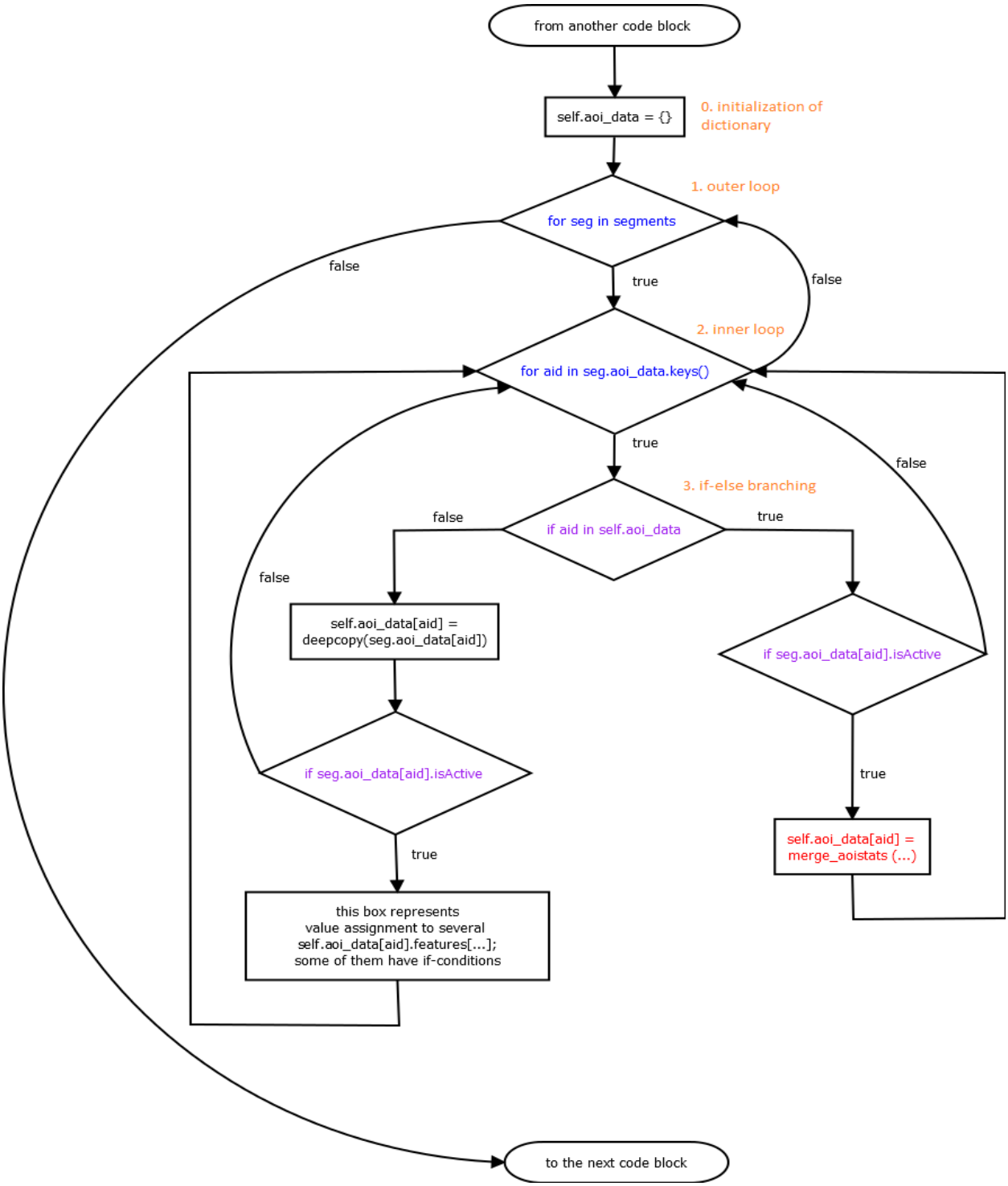
(Figure 9: an example of AOI objects as elements of aois)

Now, recall from the section Data Transformation that P18 has two segments for its only scene, main_task; in terms of code, it has the seglist [('part1', 66808, 146471), ('part2', 146471, 324262)]. The segment level loop marked in green in Figure 8 starts to run on the first element. Recall also from the same section that within the loop before the call to the constructor of Segment takes place, sublisting of the lists of data from the TobiiV3 object occurs, with the resulting lists to be passed to the initialization method of Segment for aggregation. In particular, among those lists, there is the list all_data. The start and ends times specific to the Segment object, which are used in AOI aggregation as mentioned in LP1 above, are the timestamp of the first element of the all_data and that of the last element, respectively.

For the segment part1, the execution process of the AOI level loop (the grey one in Figure 8) may be described at a high level as follows. Since the first AOI is the first of the two special cases mentioned in LP2, there is no delimiting with respect to time. After delimiting with respect to space, computation of aggregated attribute values occurs. The returned AOI object is stored in the dictionary attribute of the Segment object—self.aoid_data[‘1’]—as in LP4. Since the second AOI is the more general case stated in LP2, delimiting with respect to time takes place as well. In this case, as it turns out, there is intersection of the time interval with the segment specific interval determined by the start and end times, so that subsequent delimiting with respect to space and then computation of aggregated attribute values occur. Tthe returned AOI object is stored in self.aoi_data[‘2’]. The third AOI with timeseq: [(-1, -1)] is the second special case (always inactive) mentioned in LP2, after default attribute value assignment, the call returns; the AOI object is stored in self.aoi_data[‘3’]. The processing of the last AOI is similar to the second one since in this case, as it happens, the second interval in the list value of timeseq has an intersection.

Next, the segment level loop (green) in Figure 8 runs on ('part2', 146471, 324262), the second element in the seglist given in the penultimate paragraph. AOI level looping similar to the one described above repeats, with different segment start and end times. By the time the AOI level looping ends, the Segment for part2 also has its dictionary attribute aoi_data filled with newly created AOI objects.

This second iteration completes the segment level looping for MetroQuest P18. However, as in Figure 8, before the iteration of the higher, scene level looping (marked in blue in Figure 8) ends, there is a call to another method, set_aois, where the scene level AOI aggregation on the two Segment objects takes place. In particular, the method contains a rather complex nested loop with multiple conditional statements that can be represented in terms of a flowchart as follows:



(Figure 10: scene-level AOI aggregation loop logic in set_aois)

Given the two Segment objects for part 1 and part2, each of which stores aggregate AOI data in the aoid_data attribute of the type dictionary, the looping process can be summarized as follows.

When starting to run on the segment part 1:

0. The new attribute of the type dictionary, self.aoi_data, is created and initialized in the Scene object, representing the scene main_task, for storing the aggregated AOI data. (These scene level data, not the segment level ones, are exported later in the EMDAT generated file.)
1. The outer for-loop runs over the two Segment objects, first on that for part1.
2. The inner loop starts to run over the keys of the Segmnet object's aoi_data dictionary, namely '1', '2', '3', and '4'.
3. Since for the first iteration of the outer for-loop, the Scene obejct's aoi_data is initially empty and there is no duplication in the Segment object's aid's, i.e, dictionary keys, all iterations of the inner for-loop follow the left branch. In the process represented by the box right after the branching, the AOI objects that contain the segment level AOI aggregation data and that are stored as the values of the Segment's aoi_data dictionary are copied to the Scene object's aoi_data dictionary in each iteration of the inner loop, under the corresponding key, i.e., aid. What happens subsequently depends on the Boolean attribute isActive determined by the method call to is_active mentioned in LP2 above: if true, some of the attribute values of the AOI object just copied to the Scene object may be overwritten; otherwise, the execution proceeds to the next iteration of the inner for-loop.

When the first inner looping ends:

1. The outer-loop runs now on the segment part 2.
2. Since the keys of the aoi_data dictionary of the Segment object for part 2 are the same as those of the preceding, the inner for-loop again runs over '1', '2', '3', and '4'.
3. Because values have been assigned to those dictionary keys during the first iteration of the outer for-loop, this time all iterations of the inner for-loop follow the right branch. What happens next also depends on the Boolean attribute isActive mentioned above: if true, attribute values of the AOI object in the Segment object's aoi_data dictionary is combined to the corresponding values of the AOI object in the Scene object's aoi_data dictionary, i.e., in the preceding Segment object's in this case; otherwise, the AOI object is ignored, and the execution proceeds to the next iteration of the inner for-loop.

Common developer tasks

Adding a new eye-tracker data format. Support for new eye-tracker data format can be added by instantiating the Recording class in the EMDAT_eyetracker folder. The subclass of Recording are therefore meant to parse the input eye-tracking data. This folder already includes parsers for Tobii Studio and SMI BeGaze. The best approach is to duplicate one of this parser and modify the parsing function as needed, namely:

- read_all_data(): parse the raw gaze sample
- read_fixation_data(): parse the fixations
- read_saccade_data(): parse the saccade

- read_event_data(): parse the mouse/keyboard data

As mentioned in the User Manual only the gaze samples and the fixations are required (e.g., implementing the saccade and event parser is optional).

As shown in the exisint Tobii Studio and SMI parsers in EMDAT_eyetracker, the gaze samples, fixations, saccades and events are stored as a list of dictionaries.

Once the parser is implemented, there are two more steps to enable it:

- In params.py, add a new EYETRACKERTYPE for the new data format, and comment out the other ones
- In the BasicParticipant class (e.g., BasicParticipant.py and BasicParticipant_multiprocessing.py in the repo), modify the __init__() function to call the new parser. This is done by adding a new test in the following part of the code:

```
"""
Type of eye tracker that generated the raw data. Must be specified in params.py, so
appropriate parser is selected
"""
if params.EYETRACKERTYPE == "TobiiV2":
                rec = TobiiV2Recording(datafile, fixfile, event_file=eventfile,
                                       media_offset=params.MEDIA_OFFSET)
elif params.EYETRACKERTYPE == "TobiiV3":
                rec = TobiiV3Recording(datafile, fixfile, saccade_file=saccfile,
                                       event_file=eventfile,
                                       media_offset=params.MEDIA_OFFSET)
elif params.EYETRACKERTYPE == "SMI":
                rec = SMIRecording(datafile, fixfile, saccade_file=saccfile,
                                       event_file=eventfile,
                                       media_offset=params.MEDIA_OFFSET)
```

Defining new features. AOI-based features are defined in the AOI_Stat class in AOI.py, and non-AOI-based features are defined in the Segment class in Segment.py. Both class have access to all of the input data (gaze samples, fixations, saccades and events). It is recommended to look at the source code to see how other features are defined, to understand how this is done, and possibly duplicate an existing feature to implement a new one.
The trickier part is aggregating the features in the Scene class in scene.py. EMDAT outputs features at the scene levels, but because a scene is made of a set of segment, it is required to merge/aggregate the features of all segments that compose the scene. This aggregation is performed in different function in scene.py depending on the nature of the feature. Namely:

- Fixation-based features are aggregated in merge_fixation_features()
- Saccade features are aggregated in merge_saccade_data()
- Angles features are aggregated in merge_path_angle_features()
- Pupil features are aggregated in merge_pupil_features()
- Head distance features are aggregated in merge_distance_data()

- Interaction event (mouse, keyboard) are aggregated in merge_event_data()
- AOI-based features are aggregated in merge_aoistats(), which again call AOI-feature-specific functions (e.g., merge_aoi_fixations() for fixation-based AOI-based features).
- AOI sequence features are aggregated in merge_aoisequences()
- Blink features are aggregated in merge_blink_features()

Feature are stored in a local dictionary called `self.features` in the Scene, Segment and AOI_Stat classes. For instance the numfixation features is stored as follows:

- self.features['meanfixationduration']

It is important to name the new feature in a meaningful way to that the code remains readable. Once the feature is implemented, its name must be added to the list of features in params.py. Then EMDAT will automatically compute and export that feature.